Medusa Attack: Exploring Security Hazards of In-App QR Code Scanning

Xing Han, Yuheng Zhang, and Xue Zhang, *University of Electronic Science and Technology of China and Shanghai Qi Zhi Institute;* Zeyuan Chen, *G.O.S.S.I.P;* Mingzhe Wang, *Xidian University;* Yiwei Zhang, *Purdue University;* Siqi Ma, *The University of New South Wales;* Yu Yu, *Shanghai Qi Zhi Institute and Shanghai Jiao Tong University;* Elisa Bertino, *Purdue University;* Juanru Li, *Shanghai Qi Zhi Institute and Shanghai Jiao Tong University*

https://www.usenix.org/conference/usenixsecurity23/presentation/han-xing

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Medusa Attack: Exploring Security Hazards of In-App QR Code Scanning

Xing Han[1,2,*], Yuheng Zhang[1,2,*], Xue Zhang[1,2], Zeyuan Chen[3], Mingzhe Wang[4]
Yiwei Zhang[5], Siqi Ma[6], Yu Yu[2,7], Elisa Bertino[5], Juanru Li[2,7]
[1]*University of Electronic Science and Technology of China,* [2]*Shanghai Qi Zhi Institute,*
[3]*G.O.S.S.I.P,* [4]*Xidian University,* [5]*Purdue University,*
[6]*The University of New South Wales,* [7]*Shanghai Jiao Tong University*

Abstract

Smartphone users are eliminating traditional QR codes as many apps have integrated QR code scanning as a built-in functionality. With the support of embedded QR code scanning components, apps can read QR codes and immediately execute relevant activities, such as boarding a flight. Handling QR codes in such an automated manner is obviously user-friendly. However, this automation also creates an opportunity for attackers to exploit apps through malicious QR codes if the apps fail to properly check these codes.

In this paper, we systematize and contextualize attacks on mobile apps that use built-in QR code readers. We label these as MEDUSA attacks, which allow attackers to remotely exploit the in-app QR code scanning of a mobile app. Through a MEDUSA attack, remote attackers can invoke a specific type of app functions – *Remotely Accessible Handlers* (RAHs), and perform tasks such as sending authentication tokens or making a payment. We conducted an empirical study on 800 very popular Android and iOS apps with billions of users in the two largest mobile ecosystems, the US and mainland China mobile markets, to investigate the prevalence and severity of MEDUSA attack related security vulnerabilities. Based on our proposed vulnerability detection technique, we thoroughly examined the target apps and discovered that a wide range of them are affected. Among the 377/800 apps with in-app QR code scanning functionality, we found 123 apps containing 2,872 custom RAHs that were vulnerable to the MEDUSA attack. By constructing proof-of-concept exploits to test the severity, we confirmed 46 apps with critical or high-severity vulnerabilities, which allows attackers to access sensitive local resources or remotely modify the user data.

1 Introduction

As smartphones have become more and more ubiquitous, Quick Response (QR) codes are widely used for convenience and efficiency. According to a study by Juniper Research [36], over one billion smartphones were embedded with QR code scanning in 2022. Scanning a QR code is often the fastest and most convenient way for smartphone users to obtain or display text information in a contactless manner. Since their initial use for opening URLs to display vaccine certificates during the COVID-19 pandemic, manufacturers have started utilizing QR code scanning for multiple purposes. To further facilitate users, a new trend in QR code applications, in-app QR code scanning, has emerged. When a mobile app needs to acquire information from a QR code, instead of switching to a third-party QR code scanner app (e.g., the system camera app), the app directly uses a built-in QR code reader component to extract the encoded data. More importantly, once the encoded data is extracted, the app can automatically execute a series of functions (e.g., redeem a coupon and use it to make a payment) without any additional interactions, and thus provide a seamless user experience.

While in-app QR code scanning significantly facilitates users in many application scenarios, it poses additional security threats to mobile apps. QR codes have been previously suggested as an attack vector for downloading malware, visiting some phishing websites [41], or injecting malicious code into vulnerable web components [54]. Corresponding effective defenses have been proposed based on filtering. We observe that past works have not largely addressed the overall ecosystem of in-app QR code scanning, focusing more generically on web-to-app attacks instead. In this work, we spotlight the systematic vulnerabilities of in-app QR code scanning (i.e., mobile apps that use built-in QR readers other than the system's default one) and detail how the vector is analyzed. Specifically, we characterize such threats and attack vector against in-app QR code scanning and investigate how flaws of built-in QR code readers can be exploited to attack their host apps. We present the MEDUSA attack, an attack demonstrating that built-in QR code readers can be easily hijacked to execute native functions of the host app. In a typical MEDUSA attack, a carefully crafted QR code drives the built-in QR code reader to trigger web access with tampered parameters; then the created in-app browser component is

[*] The first two authors contributed equally to this work

hijacked to communicate with an attacker-controlled remote server. Finally, the attacker leverages a class of app functions designed to work with remote servers, referred to as *Remotely Accessible Handlers* (RAHs) in this paper, to execute the attack. In particular, the MEDUSA attack uses the RAHs customized by app developers (regarded as custom RAHs), and registers to the in-app browser component afterward, as those RAHs are more likely to perform privileged operations, such as sending sensitive data to the server or executing unexpected functionalities in the victim app.

The MEDUSA attack is different from many web-to-app or app-to-app attacks [8, 11, 22, 23, 29, 30, 47, 48, 54]. On the one hand, it aims to tamper with QR code scanning, a frequently used in-app operation for input data collection. Thus, it does not rely on any other third-party app to input the attack payloads (e.g., a crafted URI in an email app to be clicked, or an Intent sent by a malicious app). On the other hand, unlike those injection attacks that insert well-crafted malicious code (via different input sources) into legitimate web pages, our MEDUSA attack leverages the built-in QR code reader to launch a hijacked in-app browsing. Once the in-app browsing is controlled, the attack reuses the custom RAHs present in the app. In this scenario, even though Android and iOS apps (especially the popular ones) have protections against most attacks from the external (e.g., W2AI attacks [8]), their built-in QR code readers become the prevalent internal attack surface. Even worse, a MEDUSA attack often affects both Android and iOS apps because they share similar in-app QR code scanning mechanisms, and thus the flaws. An exploit can thus be reused to attack Android and iOS versions of the same app with minor modifications.

To systematically investigate how widespread the MEDUSA attack related security vulnerabilities are in Android and iOS mobile apps, we have designed a four-stage vulnerability detection approach. First of all, to enhance detection efficiency, we statically scan the code of tested apps to select those with custom RAHs as candidates. Next, we collect the official QR codes of the candidates as test cases, examining whether the built-in QR code readers use an in-app browser component (i.e., WebView) to access web content. If a WebView is used, we further analyze which custom RAHs are registered to this object. Then, we employ two additionally generated QR codes to identify insecure QR code readers. Finally, through a malicious QR code, we test whether an app can be hijacked and whether its custom RAHs are invoked to confirm the security hazards.

We applied our detection to 800 most popular Android and iOS apps in the US and mainland China by October 2022 to assess the impact of the MEDUSA attack. We found that protection against the QR code input was relatively weak. Built-in QR code readers were widely used but they often ignored the risk that the QR code could be a malicious input, and just deployed simple and ineffective sanitization. Among the 800 apps, nearly half of them (377, 47.1%) integrated built-in QR code readers, and 286 apps allowed the QR code readers to launch the in-app browser component. Nonetheless, 115 apps did not apply any defense against the scanned QR codes (and the following accessed web contents), hence allowing custom RAHs to be invoked arbitrarily. What is worse, even those apps that checked their input QR codes, some could still be easily circumvented; in our test, eight apps with more than 10 million users contained QR code readers with ill-implemented sanitization and thus were still vulnerable to the MEDUSA attack. Our analysis results show that weak protection against QR codes represents a serious vulnerability, that can be exploited by attacks, like the MEDUSA attack, affecting billions of mobile users.

Contributions. We make the following contributions:

- *New Vulnerabilities in QR Code Scanning.* We unveil new security risks in built-in QR code readers and detail the MEDUSA attack that exploits (custom) RAHs to remotely attack mobile apps. We also discuss the root cause of the MEDUSA attack and why defenses are not easy to deploy.
- *Effective Analysis Techniques.* To help detect mobile apps that are vulnerable to the MEDUSA attack at a large scale, we developed a series of novel analysis techniques to identify risky implementations in an automatic manner. Our approach to automatic vulnerability detection is organized into two independent tasks: discovering potentially exploitable apps and identifying insecurely used QR code readers. The results are then combined to determine whether an app is vulnerable.
- *Cross-platform Investigation.* Unlike many previous works that only analyzed Android apps, we extensively analyzed both Android and iOS apps and provided a meaningful snapshot of the ecosystem. We analyzed the top-200 popular Android/iOS apps of the US app market (each from Google Play and Apple AppStore, respectively) and the top-200 Android/iOS apps of the mainland China app market (each from Tencent MyAPP and Apple AppStore, respectively). Our investigation shows that there is a large number of apps that could be affected by the MEDUSA attack in both Android and iOS ecosystems.

Ethical Considerations. The MEDUSA attack related experiments were conducted using the app accounts of the authors, hence did not affect other users. All the attacks were tested on our own devices with our test accounts, which did not harm any of the victim app servers. We legally downloaded all tested apps using smartphones without any crawlers.

The goal of using code reverse engineering as part of our app analysis is not to replicate or modify legitimate apps, but to examine their implementation and find security flaws. After finding vulnerable apps, we informed the corresponding

developers as well as the organizations responsible for maintaining vulnerability databases. In the meantime, we provided solutions to help them fix the vulnerabilities we have detected.
Availability. To ensure the reproducibility of our work and to help the community evaluate future attacks and defenses, we released our datasets and analysis tools at `https://medusa.code-analysis.org`.

2 Background

2.1 QR Code

The Quick Response (QR) code is a barcode that appears as a square pattern and stores encoded data. QR codes were first created in 1994 by Denso Wave, a subsidiary of the Japanese firm Toyota Group. With the extensive use of smart Android and iOS phones, QR codes are today widely posted both online and offline. They are usually used to store ASCII text but can also store binary data. The encoding schemes of QR codes are regulated by ISO/IEC 18004:2015 [28]. For a QR code (Version 40), its maximum data encoding capacity is 23,648 bits [25]. This makes it possible to generate QR codes containing a rich amount of information. Moreover, novel compression technology including compressed data direct computing [56] can allow QR Code to contain more information. Actually, the ISO standard only defines how to encode the data but does not specify restrictions on the kind of encoded content. That is, a QR code could encode a very complex URL, an X.509 certificate, a JSON-formatted text containing a user name and a phone number, or a base64-encoded binary data of an image.

For a human user, a QR code is incomprehensible, and thus a QR code reader (often a mobile app) is necessary to scan and parse it. Both Android and iOS provide QR code reader as a system component (i.e., a system app). Nonetheless, those QR code readers only decode the QR code but do not "understand" the decoded data. Oftentimes, the decoded data is sent to another app for handling. For instance, if the decoded data is a URL, the QR code reader would launch the system default web browser to visit it (with a prompt to ask for the user consent).

2.2 In-App QR Code Scanning

Even though most smartphones provide QR code readers as pre-installed system apps, more and more apps today integrate a QR code reader as a built-in component to handle their own QR codes. Specifically, a built-in QR code reader consists of three parts (see Figure 1): a *QR code scanner* to scan and decode QR codes; a *QR code parser* to parse the decoded data (often with proprietary formats); and a *QR code executor* to automatically execute certain activities driven by the content of the decoded data. Take WhatsApp [53] as an example. When a user tries to log into her WhatsApp account on a desktop browser, the website of WhatsApp will show a QR code instead of asking the user to provide the username and password. The user would then utilize her (already logged in) WhatsApp app to scan the QR code, and then the authentication on the desktop browser is automatically executed. In this scenario, the built-in QR code reader of WhatsApp first decodes the QR code using its QR code scanner (as other third-party QR code reader apps do). It then utilizes its QR code parser to handle the (base64-encoded binary) data and send it to the QR code executor. According to what the data content indicates, the QR code executor constructs and issues a login request with the user credentials to the WhatsApp server - in essence providing a password-less authentication. Finally, the QR code executor would display information about the operation - in this case the login result instead of the decoded text of the QR code.

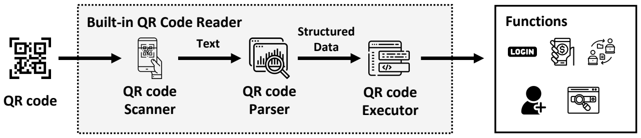

Figure 1: Components of a built-in QR code reader

Although apps (and their built-in QR code readers) follow the ISO standard to decode QR codes, they handle the decoded data in very diverse ways. It is the content of the decoded data that determines how the QR code executor would work. The use of a built-in QR code reader has two important advantages: (1) Seamless user experience: users do not need to switch to a third-party app to scan QR codes. The app can directly execute activities such as checking into a location or connecting to a password-protected wireless network without requiring any additional user interaction once the specific QR code is scanned. (2) Secure scanning: the QR code readers often adopt extra content validation to ensure that the app would not access and handle an illegal QR code.

2.3 Remotely Accessible Handlers

To facilitate interactions with their app servers, many apps explicitly expose some functions that can be invoked remotely. In this paper, we refer to such a function as a *Remotely Accessible Handler* (RAH). A RAH is often a native function, written in Java/Objective-C/C/C++, of the app. It first registers a RAH interface as a web API and then remote servers can invoke it (with some parameters) to perform further operations. Therefore, a RAH builds a web-app bridge for applications/scripts running on remote web servers to access the native app functions and resources on the mobile phone.

In Android and iOS apps, RAHs can be classified as standard and custom ones. Standard RAHs are generally web APIs [26] implemented by the official in-app web browser component (e.g., Android `WebView` [5] or iOS `WKWebView` [27]). For instance, the `Credential Management`

API allows a website to store credentials (e.g., private keys) on the client side or retrieve them from the client. Additionally, app developers also develop custom RAHs to implement app-specific functions (e.g., take a photo and upload it). To enable a custom RAH, an app first explicitly registers the RAH interface to the in-app web browser component[1]. When the app accesses web contents, it first creates a WebView object and then uses it to open a certain URL. If the custom RAHs are registered to this WebView object, then the corresponding remote server can access the RAHs through the interfaces. In the following, we detail how custom RAHs are developed and used in Android and iOS apps, respectively.

2.3.1 Custom RAHs in Android Apps

There are generally two types of custom RAHs in Android apps:
1) *RAH_a* Type-1 (**RaT-1**). This type of RAH for Android relies on `addJavascriptInterface` to register itself as a Java object into the JavaScript runtime in the Android `WebView`. The JavaScript executing in the `WebView` can then access the registered RAH object, invoking its methods with `@JavascriptInterface` annotation since Android 4.2, or all methods before Android 4.2.
2) *RAH_a* Type-2 (**RaT-2**). This type of RAH is implemented as an event handler for the Android `WebView`. By overriding the methods of `WebViewClient`, a member of the `WebView` object, an app registers event handlers as **RaT-2** RAHs that process various events generated by `WebView`.

2.3.2 Custom RAHs in iOS Apps

When using `WKWebView`, there are several ways to register and use custom RAHs.
1) *RAH_i* Type-1 (**RiT-1**). This type of RAH relies on `WKScriptMessageHandler`, an official method provided by the `WebKit` framework as the web-app bridge between an app and the `WKWebView`; by using `WKScriptMessageHandler`, a message can be sent from the `WKWebView` via the `WebKit` framework to the app and handled by the native functions of the app.
2) *RAH_i* Type-2 (**RiT-2**). WebViewJavascriptBridge [51], the most popular third-party library to create a bridge between an iOS app and `WKWebView`, has been used by a range of companies and projects such as Facebook Messenger. A **RiT-2** can use the WebViewJavascriptBridge to create web-app bridge objects in both the app and WebView Native methods can be registered as handlers and be invoked from the `WKWebView` via the bridge object.
3) *RAH_i* Type-3 (**RiT-3**). DSBridge is another well-known third-party library to invoke native methods from the `WKWebView`. By utilizing DSBridge, API objects with native methods can be added to `DWKWebView`, the wrapper for `WkWebView`, and these native methods can be invoked from the WebView by including the JavaScript library of `DSBridge`.

3 Medusa Attack

3.1 Attack Overview

Briefly, a MEDUSA attack starts when an app unintentionally scans a (malicious) QR code by using its built-in QR code reader. Then the app is tricked into executing some unexpected functions (e.g., sending credential tokens to a remote server without the permit of the user). Underneath this procedure, it is the QR code executor that creates a WebView object with some RAHs registered. The WebView object is originally created to help fulfill essential app-to-web communication (e.g., implementing user authentication), and its RAHs are expected to be invoked only by trusted remote servers. However, a meticulously crafted QR code is able to hijack the QR code executor, creating a WebView object with tampered parameters. This leads the WebView object to access some attacker-controlled resources, and the attacker can therefore remotely invoke the RAHs belonging to the WebView. In this way, a MEDUSA attack is executed. In addition, a large number of mobile apps allow their QR code executor to execute after the QR code scanning without any user interaction. Although this design is user-friendly, once the QR code executor is hijacked, the MEDUSA attack can be executed stealthily and automatically.

3.2 Attack Model

Attack Vector. The MEDUSA attack is initialized by a QR code. The attack vector is the built-in QR code reader of a mobile (either Android or iOS) app. The QR code can be dynamically generated or prepared beforehand. Once the victim app scans the malicious QR code, the door to the MEDUSA attack is opened.

In order to prepare the malicious QR code, the attacker first identifies the apps containing the (vulnerable) built-in QR code reader, and then prepares the corresponding malicious QR codes beforehand, waiting for the victim to scan them in a certain application scenario. A specific application scenario for QR codes is offline usage: many QR codes are published publicly by vendors for various purposes (e.g., advertising, making payments). Such publicly available QR codes can easily be tampered. Also, unlike text-based information (e.g., a URL) that can be more easily recognized by humans, QR codes are generally unreadable and thus it is unlikely a human can distinguish a malicious QR code from a benign one.

[1] In this paper we focus on **WebView object**, the most widely used in-app web browser component. We mainly focus on Android `WebView` and iOS `WKWebView`. Note that the `UIWebView` in iOS has been deprecated and is not recommended by Apple, and from the end of 2020, AppStore no longer accepts app updates containing `UIWebView` [49]. Thus in this paper we only discuss `WKWebView`.

Attack Constraints. The MEDUSA attack only assumes that the victim app has a built-in QR code reader and some RAHs, and the app is able to access the Internet. Apart from this, *no special permission* is required for the app. Note that the MEDUSA attack neither assumes that the target app contains any malicious third-party libraries nor needs to install any malicious apps or certificates.

The MEDUSA attack assumes that the victim apps are well protected by the mobile OS, and the integrity of the mobile device (OS and the internal storage data) is guaranteed. The MEDUSA attack also assumes that typical code and network-level defenses have been deployed. For instance, network traffic is protected by HTTPS. The only capability the attacker needs is to obtain the executable code of the target apps and employ reverse engineering to understand the logic of QR code scanning and RAH invocation. Obtaining the executable code is very easy despite the protections mentioned earlier.

3.3 Attack Steps

To construct a MEDUSA attack three steps are required in general: 1) triggering the QR code executor with a (malicious) QR code to launch web access (i.e., create a WebView object); 2) tampering with the parameters of WebView initialization using the crafted data in the QR code; and 3) hijacking web access to attacker-controlled resources and invoking RAHs. In the following, we use the SHAREit [42] file transfer app with more than 2 billion users worldwide (SHAREit for short) as a concrete example to illustrate each step, and discuss the technical challenges unique to the MEDUSA attack.

3.3.1 Web Access Triggering

The most standard approach used by Android and iOS apps to execute in-app web browsing is to create a WebView object, and leverage the object as an in-app browser component to access the web resources. Hence in this paper, we focus on the case where an app uses WebView instead of a third-party browser to load web contents. Take the SHAREit as an example; it utilizes the QR code to help users share files online and offline. If a user wants to upload files from her mobile device to her PC, she can first use the PC browser to access `http://web.ushareit.com/` and obtain a QR code on the web page, and then scan the QR code[2]. After that, the built-in QR code reader of SHAREit extracts the essential data (in this case the `cid` value), and then accesses the server of SHAREit, sends the files to the receiver designated by the `cid` value. This procedure is also a common practice for many apps: extracting data from the QR code, launching web access with the obtained data as parameters, and finally establishing communications between an app and its remote (web) server.

3.3.2 Parameter Tampering

If a WebView object is created by the QR code executor, its initialization parameters (if not hard-coded) are usually obtained from the QR code. The most common initialization parameter is the URL parameter, which indicates the content to be loaded into the WebView. Moreover, information about a certain identity (e.g., a string that represents the user/client id of SHAREit) is frequently contained in the web request. Since the QR code standard lacks an integrity verification mechanism, any content in a QR code can be easily tampered. Therefore the QR code executor that scans a tampered QR code would initialize the WebView object with modified parameters. For example, in the case of SHAREit, if we replace the domain name in the QR code (i.e., `ushareit.com`) with a malicious one (e.g., `medusa-attack.com`), the WebView object would load the replaced URL if the QR code reader does not carefully examine the input.

In response to parameter tampering, many QR code readers do add a filter in their QR code parsers. A typical scenario for many apps is to set a URL access allowlist (e.g., only allowing to access `https://*.twitter.com/*`) if the QR code encodes a URL. When such a content filter is deployed, the QR code executor would directly prompt a warning when an illegal QR code is scanned. However, this defense only blocks the most trivial tampering (i.e., directly replacing the content of the original QR code with a malicious URL); there are many complex cases in which the attackers are able to control other parameters and thus launch the MEDUSA attack. In addition, the filter cannot distinguish the malicious URL accurately (Section 5.7 and Section 6 give some examples of ineffective URL filters).

3.3.3 RAH Invoking

Once we have a QR code to create a WebView object and launch a hijacked web access, the final step is to invoke RAHs in the app to execute different functions and carry out attacking tasks. As mentioned in Section 2.3, a remote party can invoke both standard RAHs and custom RAHs. Standard RAHs are open APIs and seldom represent substantive threats because the risks of standard RAH abuse have already been pointed and corresponding defense strategies have been devised [22–24, 54]. Hence a MEDUSA attack mainly utilizes the custom RAHs registered to the created WebView object.

Each time when a WebView object is initialized, we can dynamically analyze its loading events to record its dynamically registered custom RAHs. The dynamic analysis, however, only collects the interface information of the registered custom RAHs, but does not tell how a specific RAH is invoked. Hence we need to further analyze the concrete implementation of an RAH to invoke it correctly. Take the `getSzUserInfo` RAH in SHAREit as an example; we first dynamically pinpoint the created WebView, and then obtain the name of the RAH host object by parsing the

[2] The QR code is in the form of `http://ushareit.com/device?t=2&cid=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`.

code: `this.Q.addJavascriptInterface(new WebClient(), "client")`. Next, we traverse the `client` object to find interfaces of all custom RAHs, and then we can invoke each custom RAH according to its interface declaration. In this case, the interface is `public String getSzUserInfo()`; thus we can directly invoke it with no parameter and expect to get the execution result as a string.

4 Vulnerable App Detection

In this section, we discuss how to determine whether an Android or iOS app is vulnerable to the MEDUSA attack. Note that since the QR code readers in Android and iOS apps have a similar behavior from the user end, we adopt a unified method to test apps on both platforms.

4.1 Detection Workflow

We propose a workflow for vulnerable app detection based on an analysis with four stages, namely (see Figure 2) *RAH Presence Analysis* (stage 1), *RAH Registration Analysis* (stage 2), *Insecure QR Code Reader Identification* (stage 3), and *Exploit Construction* (stage 4), described in what follows.

- **Stage 1:** The detection workflow starts from checking whether a tested app contains any custom RAHs. We observe that any custom RAH must first register itself to a host WebView object before being used. The ways according to which an Android or iOS app can register custom RAHs are limited (i.e., by invoking certain RAH interface registration functions such as `addJavascriptInterface`). Hence in this stage, each app is statically analyzed to check whether it contains code snippets of custom RAH registration to determine whether the app uses custom RAHs.
- **Stage 2:** If an app contains custom RAHs, our detection workflow then collects all custom RAHs registered to the built-in QR code executor created WebView objects, and analyzes the RAH registration procedure to understand how those RAHs should be invoked remotely. For each tested app, we manually collect its official QR code, and scan the QR code using the built-in QR code reader, respectively. Meanwhile, the execution of the app is dynamically monitored to find whether a WebView object is created. If a WebView object is created, our tool analyzes how the involved custom RAHs are invoked by extracting the custom RAH interface registration procedures.
- **Stage 3:** Once we have pinpointed the registered custom RAHs, our detection identifies whether a built-in QR code reader thoroughly checks the content of its scanned QR codes. Given an app with a QR code reader, our detection utilizes three kinds of QR codes (i.e., third-party URL, official URL, and official QR code) to test whether the QR code executor would access web resources to what extent. If it is able to access any arbitrary web resources, our detection workflow reports an insecure QR reader.
- **Stage 4:** To determine whether an app, with an insecure QR reader is actually vulnerable to the MEDUSA attack and the severity of the attack, the detection workflow constructs a proof-of-concept exploit, including an attacker-controlled QR code and a malicious webpage which invokes the custom RAHs against the victim app. If the exploit can trick the app into accessing the malicious webpage and its custom RAHs are invoked by the webpage, the detection workflow confirms the app as vulnerable.

In the following, we elaborate each detection stage.

4.2 RAH Presence Analysis

We examined the development documentations to summarize the five types of custom RAHs discussed in this paper, and collected static features (i.e., RAH registration functions) to help detect every type of them. Our detection then utilizes these features and conducts a detailed RAH interface discovering process for Android and iOS apps described in what follows.

Android Apps: We implement a static RAH interface detection tool for Android apps on top of `Soot` [37]. Our tool first uses `Soot` to extract basic information from Dalvik bytecode, including the name, member variables, super class, interfaces, and methods of classes as well as the parameter types, return types, and all tags (such as Annotation and Debug Type) of methods. After that, our detection traverses all Java classes in the app to pinpoint RAH registration procedures (we detail each type of RAH registration in Section A.1 in Appendix). For **RaT-1**, our detection searches the use of `addJavascriptInterface` method for any WebView object. For **RaT-2**, our detection uses the class inheritance relationships to find all the sub-classes of `WebViewClient`, and then traverses all methods of these sub-classes to detect if any of them is a customized event handler (i.e., a custom RAH).

iOS Apps: We detect custom RAH registrations in an iOS app by simply checking the use of specific classes and methods (the details of those classes and methods related to each type of RAH registration are listed in Section A.2 in Appendix). We use `IDA` to analyze the (decrypted) executable code of an iOS app, extract all the method names from its `__objc_selrefs` section and all the class names from the `__objc_classrefs` region [3]. After the extraction, our detection checks if the `WKWebViewConfiguration` class is used and the `addScriptMessageHandler:name:` method is called for **RiT-1**; if the `WebViewJavascriptBridge` class is used and the `registerHandler:handler:` and `bridgeForWebView:` methods are called for **RiT-2**; and if the `DWKWebView` class is used and the `addJavascriptObject:namespace:` method is called for **RiT-3**.

[3] We observe that even though many iOS apps today are developed using Swift, developers can still register custom RAHs via Objective-C. Hence our detection does not need to consider additional reverse engineering techniques, even if an app is mainly developed in Swift.

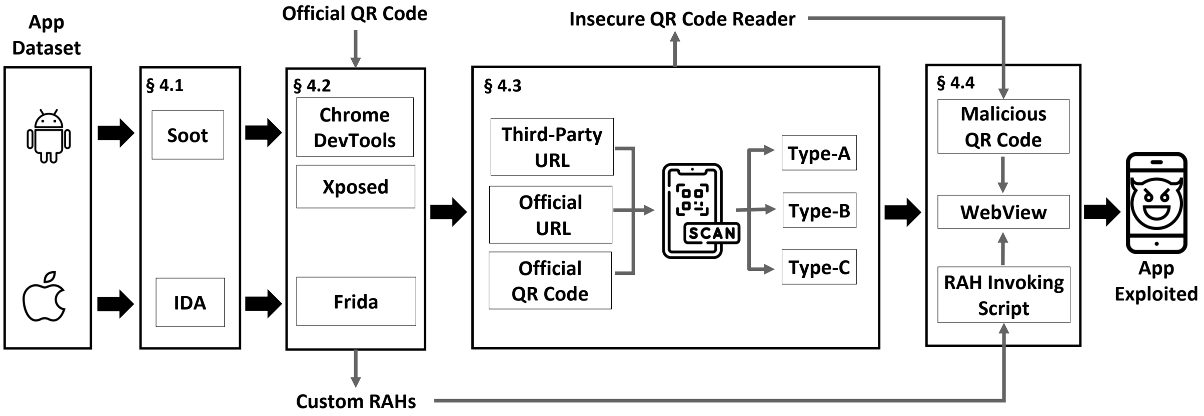

Figure 2: Workflow for identifying apps affected by the MEDUSA attack

4.3 RAH Registration Analysis

In this stage, a dynamic analysis is used to collect the registered custom RAHs when a tested app creates a WebView object after scanning the official QR code. Then the RAH registration procedure is extracted to help understand how those RAHs should be used.

Android Apps: To enforce the remote debug feature, we first use Xposed [55] instrumentation framework to execute `WebView.setWebContentsDebuggingEnabled(true)` before the Android app is executed. Next, we utilize Chrome DevTools [16] to monitor the created WebView objects. Once a WebView is pinpointed, our detection tool analyzes its relevant code snippet to find the registration procedure of custom RAHs. For **RaT-1**, our tool parses the first parameter of `addJavascriptInterface(Object object, String name)` to obtain the RAH host object, and labels all methods with the `@JavascriptInterface` annotation in this host object as interfaces of custom RAHs. Moreover, we need to also record the second parameter of `addJavascriptInterface` as the name of the RAH host object, and thus we can invoke the RAH in the form of {*RAH host object name*}.{*RAH interface name*} (e.g., `jsbridge.getUserToken()`). For **RaT-2**, since the `WebViewClient` instance can be retrieved by calling `getWebViewClient` method. We leverage Xposed to obtain the `WebView` instance, retrieve the `WebViewClient` and get its class name. Then we can obtain this `WebViewClient` instance containing customized event handler methods.

iOS Apps: For iOS apps, our detection tool dynamically monitors the methods mentioned in Section 4.2. We implement three Frida [20] scripts to parse the parameters of those method invocations, and obtain information about (custom) RAH registration. For **RiT-1**, we parse the `name` parameter of `addScriptMessageHandler:name:` method; For **RiT-2** and **RiT-3**, our script obtains the current `WKWebView` instance and the `navigationDelegate` property. If the value of the property is an instance of `WebViewJavascriptBridge` or its subclass, our script invokes the method `[_base messageHandlers]` to collect all RAHs. For **RiT-3**, our script retrieves the current `WKWebView` instance and parses its class name and superclass. If the `WKWebView` is an instance of `DWKWebView` or its sub-class, we retrieve the value of its member variable `javaScriptNamespaceInterfaces` to obtain all RAHs.

4.4 Insecure QR Code Reader Identification

To test whether a QR code reader is insecure (i.e., lacking effective input sanitization), we first use it to scan a QR code that encodes a third-party URL (e.g., `medusa-attack.com`), and check if a WebView object is straightforwardly created (**Type-A** QR code executor). For a QR code reader that does not accept this QR code, we continuously use it to scan a QR code that encodes the official URL of the app. For instance, for a tested app developed by Google, we construct a QR code that encodes `google.com` and let the app scan it. If at this time the app accepts the QR code and opens a WebView correspondingly (**Type-B** QR code executor), this implies the existence of a URL filter. A typical URL filter sets an allowlist to permit the access of a URL containing/starting with/ending with a specific string (usually the official domain name of the app). However, such a filter may still accept some meticulously crafted URLs. In Section 4.5 we demonstrate how to bypass common URL filters and in Section 5.7 we give complex examples to show that real-world URL filters are not effective.

For an app that only accepts its official QR code (**Type-C** QR code executor), we decode the official QR code to check which parts of the content are used to create the WebView object. In particular, we first parse the created WebView object to obtain its loaded URL, then we check whether some contents of the official QR code are used as the entire or part of the loaded URL. If so, we try to replace the included content with a third-party URL, testing whether we can hijack the web access.

4.5 Exploit Construction

A proof-of-concept MEDUSA attack exploit consists of two parts: a malicious QR code to lead victim app to the attacker-controlled web resources, and a malicious webpage containing an RAH invoking script to utilize the exposed RAHs.

4.5.1 Malicious QR Code

The malicious QR code embeds an attacker crafted URL in its content, inducing the QR code executor to create a WebView object and load the URL. In particular, for **Type-A** QR code executor, we simply encode the URL as a QR code. For **Type-B** QR code executor, we can craft two kinds of URLs, aiming to circumvent the URL filter. The first kind of URL leverages the complexity of URL definition according to RFC-1738 [39]. Since a URL can include various possible separators, it is very challenging for developers to apply an accurate filter even if they adopt complex pattern matching. For example, if we craft a URL `http://allowlist.com:123456@medusa-attack.com/` which contains a user name of `allowlist.com` and a password `123456`, the filter may be tricked into believing that the domain name of this URL is actually `allowlist.com`. Similarly, if the filter defines a pattern of `*.allowlist.com`, we can construct a URL `http://medusa-attack.com#whitelist.com/` to bypass the filtering. The second kind of URL leverages the open redirect issue [14]. We observe that for many apps we can find a redirect proxy in their official websites (e.g., `https://app.com/redirect.php?url=https://attack.com`). Our malicious QR code leverages such redirect proxies to craft a URL with redirection, leading the WebView object to eventually visit the attack-controlled web resource. For **Type-C** QR code executor, only if we can hijack the WebView object by mutating the official QR code, we can construct a corresponding malicious QR code. Otherwise there is no circumvention method.

In a variant of MEDUSA attack, the attacker can make use of the app link (deep link [15] in Android and universal link [50] in iOS) to conduct a cross-app RAH invocation. That is, the malicious QR code contains an app link that could direct the execution from the host app to a guest app, and the guest app may also execute some RAHs according to its handler to the app link. However, in this situation there is an explicit execution redirection, which can be easily noticed by the user. Hence we leave it as a future work to explore such a variant, although it extends the range of standard MEDUSA attack.

4.5.2 RAH Invoking Script

After the malicious QR code is generated, the next task is to develop an RAH invoking script and deploy it for the app to load. However, even though we can pinpoint all available custom RAHs, it is not easy to successfully invoke them remotely. RAHs are not always invoked in a unified way, and to access some of them a series of parameters should be first prepared. Since the app does not provide information about how to invoke RAHs, we additionally examine the official websites of apps to find how their web pages invoke these custom RAHs. For instance, to invoke an already-registered custom **RiT-2** RAH, an extra step is required at the web side. As Listing 1 shows, only when the web page (JS code) first explicitly sets an attribute of `WVJBframe.src` as '`https //__bridge_loaded__`', the registered custom RAHs is actually loaded by the WebView object. Otherwise, when a `WKWebView` object is created, no RAHs is attached [4].

Listing 1 Sample JS code of `WebViewJavascriptBridge` (partial)

```
function setupWebViewJavascriptBridge(callback) {
    ...
    var WVJBIframe = document.createElement('iframe');
    WVJBIframe.style.display = 'none';
    WVJBIframe.src = 'https://__bridge_loaded__';
    document.documentElement.appendChild(WVJBIframe);
    ...
}
setupWebViewJavascriptBridge(function (bridge) {
    bridge.callHandler('nativeMethod', {'arg':'value'},
        function (res) {
            ...
        }
    )
});
```

5 Evaluation

5.1 Datasets

To evaluate whether real-world Android and iOS apps are vulnerable to the MEDUSA attack, we collected 800 top apps from Apple AppStore, Google play, and Tencent Myapp (the largest Android app market in mainland China) as four app datasets (each contains 200 apps). We chose the mobile app ecosystems of the US and mainland China as our study targets because they are the two largest ecosystems.

5.2 Experiment Setup

We utilized four mobile devices, namely a Google Pixel 2 Android phone with Android 8, a Motorola Edge 20 Pro Android phone with Android 11, an iPhone 11 with iOS version 14.0.1, an iPhone SE with iOS version 14.0, to collect and analyze the apps. We first manually installed those apps on each device, and then extracted their executable to conduct our static analysis and perform on-device dynamic analysis.

[4] This example is actually provided at GitHub [51].

Table 1: How apps use built-in QR code readers

Detaset	# of Apps	S_1	S_2	S_3
Android-CN	200	148(74.0%)	131(65.5%)	56(28%)
Android-US	200	60(30.0%)	34(17.0%)	17(8.5%)
iOS-CN	200	123(61.5%)	108(54.0%)	39(19.5%)
iOS-US	200	46(23.0%)	14(7.0%)	3(1.5%)
Total	**800**	**377(47.1%)**	**287(35.9%)**	**115(14.4%)**

S_1: Apps with built-in QR code readers
S_2: Apps that their built-in QR code readers could launch a WebView
S_3: Apps that contain insecure QR code readers

To handle iOS app encryption, we utilized FoulWrapper [19], an automated iOS app decryptor working on jailbroken iPhones, to extract executable and resource files of iOS apps. We also modified the standard AOSP source code to build an Android app unpacking framework to handle packed Android apps (especially those apps from Tencent Myapp). Our static analysis for app executable used IDA Pro and Soot and was performed on a Ubuntu 22.04 server with two Intel Xeon Platinum 8358 Processors and 2TB RAM. We released our analysis tools and results on our website [31].

5.3 Analysis of Built-in QR Code Readers

To confirm the use of built-in QR code reader in an app, we first manually checked whether it supports QR code scanning, and if so then applied our vulnerability detection process to find insecure QR code readers. Table 1 gives the detailed analysis results for the apps with built-in QR code readers. Among the 800 apps, our analysis shows that nearly half of them (377/800, 47.1%) utilize built-in QR code readers. Particularly, the ratio of Android apps using built-in QR code readers (42.2%, 169/400) is lower than that of iOS apps (52.0%, 208/400). An interesting observation is that the apps used in mainland China are more likely to use built-in QR code readers than the ones used in the US (67.7%, 271/400 vs. 26.5%, 106/400). Such results are consistent with the statistics indicating that China is using QR codes excessively as a bridge between online and offline [35].

It is important to notice that today most QR codes are specifically designed to be handled only by the corresponding apps. For instance, a QR code designed for Outlook authentication only allows the Outlook app to handle the short-lived token to sign into the accounts of the users [33]. Other apps would not be able to parse the token embedded in such a QR code. Such a uniqueness enforces one prerequisite for the MEDUSA attack, that is, that target apps should be installed to scan QR codes. However, the result is that users would have to install various mobile apps in order to scan the codes. As a result, large numbers of apps are installed that could be prone to the MEDUSA attack.

We collected the official QR codes of all the 377 apps and utilized these QR codes to test whether the built-in QR code readers launched WebView. After scanning the official QR codes, more than three-quarters of the apps with built-in QR code readers (76.1%, 287/377) would create a WebView object to access web content. To understand why apps frequently access the web after QR code scanning, we manually checked the functionalities of those built-in QR code readers. Figure 3 shows the distribution of the functionalities that access the web. We can see from the figure that only a small portion of the built-in QR code readers are used to execute the most naive show-the-content task, while other QR code readers generally implement more sophisticated tasks, such as making a payment, starting a online meeting, etc. This implies that web access is essential for the operations following the QR code scanning.

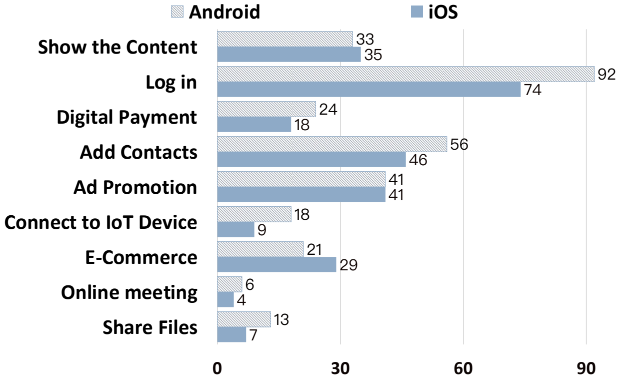

Figure 3: Distribution of the functionalities for different QR code readers

After identifying the apps whose built-in QR code readers can launch a WebView, we investigated how many of the QR code readers were able to launch a WebView to access malicious web content and the created WebView contains custom RAHs. To our surprise, we found 115 apps that created a WebView object without applying any sanitization to the input received by their QR code readers; they always accepted a QR code encoding a malicious URL (and loaded it). In addition, we discovered that even though apps would reject a QR code with a malicious URL directly encoded, they could still be flawed. By applying the two bypass approaches mentioned in Section 4.5.1, we found that eight apps with more than 10 million users were still tricked into accepting a sophisticated crafted QR code. Details about how we bypass those QR code content filters and our manual inspection of implementation flaws are discussed in Section 5.5.

Finding I: 377 out of the 800 analyzed popular mobile apps use built-in QR code readers; 123 of them (36.5%) are insecurely used.

Table 2: Numbers of apps with different types of custom RAHs registered

Dataset	APPs with RAH	RaT-1	RaT-2	Dataset	APPs with RAH	RiT-1	RiT-2	RiT-3
Android-CN	182/200 (91.0%)	180	182	iOS-CN	181/200 (90.5%)	180	109	5
Android-US	185/200 (92.5%)	171	185	iOS-US	102/200 (51.0%)	102	7	0

5.4 Analysis of Custom RAHs

In our experiments, we first conducted a scalable static analysis to filter the apps without custom RAHs. Our tool spent 1.5 on average seconds for scanning an app and reported the types of RAHs that the app contains; different apps can be analyzed in parallel. Note that such a time does not include pre-analysis costs, such as IDA database creation. Table 2 reports data about the frequency of use by the analyzed Android and iOS apps for the different types of custom RAHs. We can see that popular mobile apps commonly register custom RAHs. Even for the iOS-US dataset with the lowest ratio of RAH registrations, more than half of the apps in the dataset use at least one type of custom RAH. For each of the other three datasets, the ratio of RAH registrations exceeds 90%. Furthermore, we find that mobile apps also use multiple types of RAHs. Among the 400 Android apps, our analysis results show that 351 (87.6%) of them contain **RaT-1** and 367 (91.6%) contain **RaT-2**. In comparison, 282 (70.5%) iOS apps contain **RiT-1**, 116 (29.0%) apps contain **RiT-2**, and only 5 (1.3%) apps contain **RiT-3**. The only major difference between the US and Chinese app developers is that Chinese iOS app developers tend to use **RiT-2**, while only a few US developers use this type of RAH.

Our next experiments investigated the details of custom RAHs when a WebView is launched. We monitored the creation of a WebView and analyzed its custom RAH registration procedure to obtain information of its belonging custom RAHs (including the interface name and the name of the RAH host object). In total, our dynamic analysis discovered 2,872 custom RAHs (2,251 in Android apps, 621 in iOS apps).

Finding II: Custom RAHs are present in 650 out of the 800 analyzed popular mobile apps. The apps with insecure QR code readers contain 2,872 custom RAHs that could be exploited.

5.5 Vulnerability Exploitation

In our experiment, we first generated malicious QR codes for all 123 apps with insecure QR code readers. For those 115 apps without any URL sanitization, we simply generated a QR code with a malicious URL encoded. For the other 8 apps that required a more sophisticated QR code to bypass their content filters, we generally leveraged two types of implementation flaws. The first type of flawed implementation employed incomplete URL parsing. That is, the content filter does not consider the many corner cases of URL formats and thus our attack can utilize a URL with an irregular format to bypass the filter. In our experiment, 6 apps suffered from this flaw. The second type of flawed implementation ignored the URL redirection; the attacker could then find a URL with a legal domain name but supporting URL redirection to indirectly access the originally blocked address. In our experiment, 2 apps suffered from this flaw.

After using malicious QR codes to trigger WebView initialization, we were able to access all 2,872 custom RAHs. An interesting observation is that an app often uses similar or even the same RAHs in its Android and iOS versions. Take the apps developed by SHEIN [43] e-commerce company (43.7m+ users) as an example; we could efficiently construct one RAH invoking script to access custom RAHs in the two versions of the app. To further investigate the influence of each RAH, we manually reviewed them. If a custom RAH does not obfuscate its interface name, we can infer its importance from the name. For instance, the custom RAHs `getDeviceInfo`, `getUserAccessToken`, `nativeLogin`, and `getCurrentPosition` were judged as privileged functions. For those custom RAHs with ambiguous interface names, we checked their implementations to confirm their functionalities. Finally, our review labelled 515 custom RAHs in 80 apps (472/43 in 56/24 Android/iOS apps) as sensitive ones.

Interestingly, we found that not all sensitive custom RAHs returned execution results after having been invoked. In particular, 317 of them (61.5%) returned execution results, while the rest 213 custom RAHs did not respond. We then inspected those 213 RAHs, and found that they would execute only after a specific validation. However, the validations used by these custom RAHs are very ad hoc, and some of them may also be bypassed. It is thus clear that a general effective defense for custom RAHs does not exist; in Section 6 we further discuss the RAH protection issue.

By utilizing those sensitive custom RAHs, an attacker could manipulate the victim apps to leak sensitive user data such as authentication tokens and private information. In particular, 254 custom RAHs leaked user private information, such as contacts and location information, and 31 ones leaked authentication tokens. More seriously, we found that some custom RAHs could be used to remotely modify user data: 28 custom RAHs allowed the invoker to store data at the mobile device, and 4 custom RAHs even allowed the invoker to manipulate the victim apps to modify the user data stored at app servers. In total, **we verified that 46 apps have high-severity**

vulnerabilities.

> **Finding III:** 515 out of the 2,872 custom RAHs in 80/123 apps with insecure QR code readers execute dangerous operations, and 317 of them could be abused to carry out high-severity security/privacy attacks.

5.6 Analysis Accuracy

The accuracy of our analysis can be evaluated from three aspects. First, the identification of the 123 insecurely used QR code readers did not suffer from false positives but only from some false negatives. Although some QR code readers with more complex URL filters, labeled as "secure" in our tests, could still be circumvented by advanced exploits, we argue that currently there is no unified approach to detect vulnerabilities enabling such advanced exploit. Second, since the 2,872 custom RAHs were discovered using a dynamic analysis during the actual execution, this number is confirmed to be accurate without any false negatives and false positives. Finally, for the 317 out of the 515 dangerous custom RAHs that affected 46 apps, we tested them and the vulnerabilities were confirmed. Of course, it would be still possible to execute the other 213 RAHs out of the 515 dangerous custom RAHs. However, such an execution would have required us to monitor how the legitimate users invoke them and thus is out of the scope of our detection (for obvious privacy and legal reasons). In summary, although we cannot obtain the ground truth of our entire datasets, our detection guarantees that the reported vulnerable apps are actually affected by the MEDUSA attack.

5.7 Case Studies

5.7.1 Authentication Token Leak

Bank of China [9] is the third largest bank in China with more than 321 million registered users. Its mobile apps (both Android and iOS versions) contain built-in QR code readers able to create WebView objects. However, the web access can only be hijacked in a very indirect way. In fact, we found that neither the Android nor the iOS app accepted a third-party URL, which indicated that those apps use a filter to restrict the URL to be accessed. A more comprehensive analysis we conducted demonstrated that the apps only accepted URLs with hard-coded prefix, which means that one cannot directly tamper the URL body to hijack the apps. Nonetheless, we examined the official QR code and found a specific `functionCode?=xxx` parameter in the URL, which suggests that the app (we use the Android app as an example) utilized ARouter [7], a third-party URL parsing and re-direction framework to execute different routines according to the value of this parameter (that part of the QR code content was processed as Listing 2 shows). We observed that if the value of `functionCode` parameter equals to *crossborderWebPage*, the ARouter would launch an Activity that is able to trigger an in-app browsing. Moreover, if at that time we provided an extra `url?=https://xxx.com` parameter, the Activity would transfer the value to its created WebView object. As a result, we could craft a URL with certain `functionCode` and `url` parameters to hijack the indirectly launched WebView to access malicious web contents. This MEDUSA attack case shows that a QR code could affect different components in the app to create WebView objects and that a comprehensive input sanitization is required to eliminate every kind of malicious input.

Listing 2 Processing code of QR code content in BOC app (we use the Android version as an example)

```
if (WebUrl.getShareHomeUrl().equals(query) && ... ) {
    ...
    qRCodeModel.setRoutePath(query.replace(
        "functionCode=", "BOCBANK://search/"));
        return qRCodeModel;
    ...
}
```

5.7.2 Denial-of-Service Attack

Taobao app [46] is the official mobile app of Taobao, a popular Chinese E-Commerce website – the 8th most visited website in the world and the 5th most visited website in China as of 2021. It maintained an allowlist to restrict the URL to only access the Taobao website. However, the Taobao website is very large and inevitably contains many sub-sites. We could find some URLs with HTTP-redirection enabled to bypass the filter. After the HTTP-redirection bypassed the URL filter, it would trigger the app to show an Alert window when the actually accessed web page was loaded. To construct a more stealthy attack, we attempted to replace the redirection target from a URL to an app link [6, 15]. This however would crash the app. We examined the code to find whether the app had received the app link containing a parameter we maliciously construct. We found that the app would still starts the Activity the app link registers to and pass the parameter to it. Since the Activity fails to handle the malicious parameter, it throws an uncaught exception and crashes the host app.

5.8 Manual Effort Required

Most of the detection steps discussed in Section 4 are executed automatically. There are only three tasks that require manual operations. First, our experiments require the analysts to collect official QR codes and register for each tested app a user account. On average, it costs one author five minutes to collect the official QR code as well as register an account for each of the 377 apps with QR code readers. Note that the preparation does not need security expertise. Second, after the identification of custom RAHs, a manual analysis is needed to assess their functionalities and develop relevant scripts for invoking them. For the 2,872 custom RAHs, two of the paper authors spent five days (eight hours per day) to manually assess them. Thus, the average cost for one person to assess one custom RAH is less than two minutes. Finally, the most time-consuming part of the experiment is to construct a malicious QR code to circumvent the URL filters. One of the paper authors, with good web security skills, carried out this task and took an average of one hour to successfully generate an exploitable QR code. Fortunately, in our experiments most of the vulnerable apps (115/123) did not required such an effort due to the lack of input check. We had to conduct URL filter bypass only for eight apps.

Ideally, advanced program analysis techniques such as symbolic execution and taint analysis could automate the detection of flawed QR code scanning. In real-world scenarios, however, many issues hinder their use. First, in today's mobile apps, the detection of the MEDUSA attack vulnerability involves the analysis of different code layers (native code, Java bytecode, and HTML+JS code) and the existing taint analysis tools or symbolic execution tools cannot handle such cross-layer code well. Second, many apps apply code obfuscation and code packing to prevent a fully automated, fine-grained static analysis. Therefore, our detection intentionally utilizes a heuristic analysis instead of a heavy-weight analysis (e.g., symbolic execution or code slicing) to guarantee that it can be executed on most popular mobile apps.

6 Countermeasures

Mitigating the MEDUSA attack is not easy for both the in-app QR code scanning and the use of RAHs must consider many usability issues. In this section we discuss how to mitigate the MEDUSA attack by three typical defense strategies. We also showcase some ill-implemented defenses to demonstrate the difficulties of fixing MEDUSA attack related flaws.

QR Code Signature. Since the input added through the build-in QR reader can be manipulated by any party, such a scheme is insecure. As discussed in Section 5.5, it is difficult to correctly implement content filters correctly that only aim to process QR contents. To defend against the MEDUSA attack, signature verification could be applied to check that the QR code is generated by a trusted source. Such a mechanism could prevent the processing of maliciously constructed QR codes. For example, according to the signature verification proposed by previous works [17, 40, 45], when generating a QR code, the content is signed using cryptographic algorithms. Then the digital signature is integrated into the QR code. When scanning the QR codes, the app verifies the signature to determine whether the QR code is trusted.

WebView URL Filter. Since a MEDUSA attack mainly aims to hijack the WebView object to access malicious web contents, it is critical to implement a filter to prevent the WebView from opening untrusted URLs in the app. In Android, we can override the `shouldOverrideUrlLoading` method of WebView to strictly filter each URL before loading it. In this case, we can prevent attackers from abusing domains in the whitelist to redirect the URL from a trust one to an untrusted URL. In iOS, we can adopt the `WKNavigationDelegate` protocol and implement the methods of `ForNavigationActionPolicy`, which determine whether the WebView can navigate to the given URL. Then we can instantiate and set it to `WKWebView` and all the redirection in the WebView would be filtered.

We argue that although applying a filter before the loading the URL could block malicious input to a certain extent, in real world the apps and their scanned URLs are very diversified. If the app developer are not able to clearly understand the the root cause of the MEDUSA attack, they may design incomplete or even incorrect defenses. During our research we contacted several app developers and sent them details about vulnerabilities in their apps, trying to help them address the issues. Unfortunately, we observed that although many developers patched their apps accordingly, those patches did not fundamentally fix the vulnerability. Take the 12306 [1] app, an official app of China State Railway Group Co., Ltd. (CHINA RAILWAY) that serves 1.4 billion users, as an example. Both the Android and iOS (version 5.5.1.4, by August 2022) apps were vulnerable to the MEDUSA attack. After we submitted detailed information about the vulnerabilities to the developers, we received responses claiming that the apps had been updated and the vulnerabilities fixed. Unfortunately, our second inspection of the updated version (5.6.0.8) found that developers only had added an incomplete filter to the QR code parser. The filter extracted the domain from the QR code content in the URL format incorrectly. More specifically, it failed to resolve the '@' separator, and only checked if the domain contained some hard-coded strings. As a result, we submitted another vulnerability report to the developers again, and finally helped them deploy a secure filter.

Another case of bad fix is the SuiShenBan [44] app, an E-Identity app serving more than 30 million citizens of Shanghai, the largest city of China. During the COVID-19 pandemic the app had been widely used to scan QR codes for infection control and prevention. We found that after the app scans a QR code, it would first send the decoded data to a remote server for filtering, and if the remote server gives a positive feedback on the decoded data (i.e., the URL is accessible), the app would create a WebView object to access relevant web content. However, some domains in the allowlist still help open malicious web pages inside the app. For example, we found a web page, whose domain name is in the allowlist, but which would redirect to any other URLs if a parameter named `redirect_uri` is passed when the web page is loaded. This kind of web page is very common in the allowlist and we randomly selected one of them as a proxy, launching the MEDUSA attack and obtaining authentication ticket of users.

After we reported the flaw to the developers, they incorrectly patched the remote filtering server to only add the known proxy web pages into a denylist. As a result, attackers could easily replace the prohibited proxy with another one to launch the MEDUSA attack again. We finally helped them address this issue by suggesting to apply a second filter before the WebView loads any URL.

RAH protection. Developers can also check the identity of the invoker inside an RAH to prevent a malicious web server from using their custom RAHs. Unfortunately, like the case of the WebView URL filters, the checks inside custom RAHs are also error-prone. An example is the app of China Construction Bank [10] (one of the largest banks in China with more than 280 million users). Its `CCBBridge` RAH, which is able to access the authentication token of the user, applied a filter inside to check if it is invoked from a web page whose URL contains a "`.ccb.com`" string. We found that this filter was deployed in both Android and iOS versions. Obviously, such a filter is easy to circumvent by adding the "magic" `.ccb.com` string to the query of a malicious URL as a parameter, not affecting the web access, thus making possible to invoke the RAH successfully.

Another interesting observation is that many popular frameworks (e.g., `React Native` [38], `Cordova` [12], `Weex` [52], `Nebula` [32]) for building platform-independent mobile apps also register custom RAHs. Such frameworks often provide a unified way for developers to register their own custom RAHs, and allow users to configure access control policies for the custom RAHs. Consider the Apache Cordova framework as an example. It introduces a `__cordovaNative` object to help JavaScript in the WebView object access any native functions out of the sandbox without alert. But if the developer explicitly defines an allowlist to restrict the access range of this object [3], the attacker is unable to abuse it. Unfortunately, we did not find concrete instructions in official documents of these frameworks to guide developers on how to configure the protection against custom RAHs, As a result, many developers adopt default configurations and all relevant custom RAHs are exposed without any protection.

7 Related Work

QR Code Related Attacks. Previous works have focused on the use of QR codes and direct attacks against them. Lerner *et al.* [2] examined general use patterns of QR codes in the wild and identified common and uncommon uses and misuses. Kieseberg *et al.* [34] launched various attacks by changing some pixels of QR codes (e.g., SQL injection, phishing and social engineering). Those attacks show that QR codes can be easily modified by attackers. Hence it is essential for developers to implement suitable protection mechanisms when scanning untrusted QR codes. Averin and Zyulyarkina proposed QRGen [4], a tool for generating malicious QR code to assess whether the QR code scanners of blockchain are securely designed. Carboni *et al.* [21] proposed a fuzzing-based approach to automatically test QR code scanners. However, these works only focus on the QR codes, while the MEDUSA attack aims to hijack the in-app browsing after QR code scanning, triggering the custom RAHs that enable attackers to invoke the native functions in the apps.

Web-to-App Attacks. Past research works have identified and analyzed weaknesses of different in-app web browsing components and related web-to-app attacks. A major category of web-to-app attacks employs **code injection** against in-app web components. Luo *et al.* [47] first analyzed and classified attacks targeting WebView systematically. After that, Jin *et al.* [54] discovered 14 code injection channels (including QR code) enabling malicious code execution in HTML5-based mobile apps. Rizzo *et al.* [11] evaluated the risk of JavaScript interface abuse by mean of a man-in-the-middle (MiTM) attack to inject malicious code into Android WebView; Xiao *et al.* [18] further analyzed JavaScript-based cross-platform frameworks and revealed that they exposed many attack surfaces for malicious script injection. Jin *et al.* [57] showed that meticulously crafted inputs could be injected into real-world Electron (HTML and JS) apps. Since all these attacks rely on injected code to execute malicious behaviors, several types of defense have been deployed to protect against different attacks through different surfaces: sanitizers are used to filter malicious inputs; Content Security Policy and X-Frame-Options are used to restrict the browser to block contents from harmful sources; and enforcement of Strict Transport Security to defend against communication tampering. Mobile OSes and frameworks also have adopted additional protection against potentially harmful inputs. Take the PhoneGap attack [47] as an example; frameworks, like PhoneGap and its successor Cordova), now enforce validation against inputs from vulnerable APIs such as `barcodeScanner.scan` [13]. As a result, those code injection attacks are less feasible against today's mobile apps.

Unlike code injection attacks, the MEDUSA attack mainly **reuses** registered custom RAHs in victim apps instead of injecting new code to carry out the attack. Since the MEDUSA attack does not need to inject any new code or any extra execution flows, it is more difficult to defend against it with a system-level security policy. Similar to the MEDUSA attack, the Web-to-App Injection (W2AI) attack proposed by Hassanshahi *et al.* [8] also aimed to utilize remotely accessible interfaces. However, the W2AI attack only works against Android apps, and requires another app to send an Android Intent containing a crafted hyperlink (i.e., URI intent or Web Intent), while the MEDUSA attack covers both Android and iOS platforms and does not rely on the installation of any other third-party apps. Moreover, a Web Intent is not allowed to launch in-app web browsing (the Android OS by default invokes the system's built-in browser to handle the URI) starting from Android 12 [15], making W2AI attacks infeasible on the latest mobile devices.

8 Conclusion

In this paper, we systematically analyzed the security of modern QR code scanning functionality, and proposed a new attack, the Medusa attack, which exploits insecure built-in QR code readers to exploit mobile apps and breach user privacy. Typically, the Medusa attack utilizes meticulously crafted QR codes to trigger custom RAHs in apps and trick them into executing malicious actions. To assess whether the Medusa attack could affect large numbers of apps, we designed a four-stage detection approach to identify risky implementations among various built-in QR code readers. With an evaluation of 800 popular Android and iOS apps, we found that 123 apps were vulnerable to the Medusa attack, and 46 apps contained critical or high vulnerabilities.

Acknowledgments

The authors would like to thank our shepherd and the reviewers for their valuable feedback, and gratefully acknowledge the support from the National Natural Science Foundation of China (under Grant No.62002222). We specially thank Alibaba Group for the support of this research within the *SJTU-Alibaba Security Research Program*. We would also like to express appreciation to Sun Weiwei, Yuan Zhou, Zhang Weijie, and Yin Jun from *Waterfront Garden* who generously provided iOS devices to support our research. Juanru Li (*mail@lijuanru.com*) is the corresponding author.

References

[1] **12306 CHINA RAILWAY**. https://www.12306.cn/en/index.html. Accessed 2023.

[2] Adam Lerner and Alisha Saxena and Kirk Ouimet and Ben Turley and Anthony Vance and Tadayoshi Kohno and Franziska Roesner. **Analyzing the Use of Quick Response Codes in the Wild**. In Proc. 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15), 2015.

[3] **Allow List Guide | Apache Cordova**. https://cordova.apache.org/docs/en/11.x/guide/appdev/allowlist/index.html. Accessed 2022.

[4] Andrey Averin and Natalya Zyulyarkina. **Malicious Qr-Code Threats and Vulnerability of Blockchain**. In Proc. 2020 Global Smart Industry Conference (GloSIC), 2020.

[5] **WebView | Android Developers**. https://developer.android.com/reference/android/webkit/WebView. Accessed 2022.

[6] **Applinks | Apple Developer Documentation**. https://developer.apple.com/documentation/bundleresources/applinks. Accessed 2023.

[7] **ARouter • A Framework for Assisting in the Renovation of Android App Componentization**. https://github.com/alibaba/ARouter. Accessed 2022.

[8] Behnaz Hassanshahi and Yaoqi Jia and Roland HC Yap and Prateek Saxena and Zhenkai Liang. **Web-to-Application Injection Attacks on Android: Characterization and Detection**. In Proc. 20th European Symposium on Research in Computer Security (ESORICS), 2015.

[9] **Bank of China Global Web Site**. https://www.boc.cn/en/. Accessed 2022.

[10] **China Construction Bank Web Site**. http://en.ccb.com/en/home/indexv3.html. Accessed 2022.

[11] Claudio Rizzo and Lorenzo Cavallaro and Johannes Kinder. **Babelview: Evaluating the Impact of Code Injection Attacks in Mobile Webviews**. In Proc. 21st International Symposium on Research in Attacks, Intrusions, and Defenses (RAID), 2018.

[12] **Apache Cordova**. https://cordova.apache.org/. Accessed 2023.

[13] **Security Guide - Apache Cordova**. https://cordova.apache.org/docs/en/11.x/guide/appdev/security/#validate-all-user-input. Accessed 2023.

[14] **CWE-601: URL Redirection to Untrusted Site**. https://cwe.mitre.org/data/definitions/601.html. Accessed 2023.

[15] **Create Deep Links to App Content | Android Developers**. https://developer.android.com/training/app-links/deep-linking. Accessed 2022.

[16] **Chorme DevTools | Chrome Developers**. https://developer.chrome.com/docs/devtools/. Accessed 2022.

[17] Faisal Razzak. **Spamming the Internet of Things: A Possibility and its probable Solution**. In Proc. Procedia Computer Science, 2012.

[18] Feng Xiao and Zheng Yang and Joey Allen and Guangliang Yang and Grant Williams and Wenke Lee. **Understanding and Mitigating Remote Code Execution Vulnerabilities in Cross-platform Ecosystem**.

In Proc. 29th ACM SIGSAC Conference on Computer and Communications Security (CCS), 2022.

[19] **FoulDecrypt**. https://github.com/Lessica/fouldecrypt. Accessed 2022.

[20] **Frida • A world-class dynamic instrumentation framework**. https://frida.re/. Accessed 2022.

[21] **QR code Fuzzer Testing Toolkit for Android & iOS**. https://github.com/Maxelweb/FuzzQR. Accessed 2022.

[22] Guangliang Yang and Jeff Huang and Guofei Gu. **Automated Generation of Event-Oriented Exploits in Android Hybrid Apps**. In Proc. 25th Network and Distributed System Security Symposium (NDSS), 2018.

[23] Guangliang Yang and Jeff Huang and Guofei Gu. **Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities**. In Proc. 28th USENIX Security Symposium (USENIX Security), 2019.

[24] Guangliang Yang and Jeff Huang and Guofei Gu and Abner Mendoza. **Study and Mitigation of Origin Stripping Vulnerabilities in Hybrid-postMessage Enabled Mobile Applications**. In Proc. 39th IEEE Symposium on Security and Privacy (SP), 2018.

[25] **Information Capacity and Versions of the QR Code**. https://www.qrcode.com/en/about/version.html. Accessed 2022.

[26] **Introduction to Web APIs**. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction. Accessed 2022.

[27] **WKWebView | Apple Developer Documentation**. https://developer.apple.com/documentation/webkit/wkwebview?language=objc. Accessed 2022.

[28] **ISO/IEC 18004:2015**. https://www.iso.org/standard/62021.html. Accessed 2022.

[29] Lei Zhang and Zhibo Zhang and Ancong Liu and Yinzhi Cao and Xiaohan Zhang and Yanjun Chen and Yuan Zhang and Guangliang Yang and Min Yang. **Identity Confusion in WebView-based Mobile App-in-app Ecosystems**. In Proc. 31st USENIX Security Symposium (USENIX Security), 2022.

[30] Matthias Neugschwandtner and Martina Lindorfer and Christian Platzer. **A View to a Kill: WebView Exploitation**. In Proc. 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET), 2013.

[31] **Medusa Website**. https://anonymous.4open.science/r/MEDUSA_Attack-2B32. Accessed 2022.

[32] **Nebula SDK Documentation**. https://nebulasdk.alipay.com/en-us/nebulasdk/index.html. Accessed 2023.

[33] **Use a QR code to sign-in to the Outlook mobile apps**. https://learn.microsoft.com/en-us/microsoft-365/admin/manage/use-qr-code-download-outlook. Accessed 2023.

[34] Peter Kieseberg and Sebastian Schrittwieser and Manuel Leithner and Martin Mulazzani and Edgar Weippl and Lindsay Munroe and Mayank Sinha. **Malicious Pixels Using QR Codes as Attack Vector**. In Proc. Trustworthy Ubiquitous Computing, 2012.

[35] **QR Codes in China - Almost a Different Place on Earth**. https://www.qrcode-tiger.com/qr-codes-have-been-used-all-around-china-and-you-can-also-track-all-the-data. Accessed 2022.

[36] **Mobile QR Code Coupon Redemptions to Surge, Surpassing 5.3 Billion by 2022**. https://www.juniperresearch.com/press/mobile-qr-code-coupon-redemptions-to-surge. Accessed 2022.

[37] Raja Vallée-Rai and Phong Co and Etienne Gagnon and Laurie Hendren and Patrick Lam and Vijay Sundaresan. **Soot: A Java Bytecode Optimization Framework**. In Proc. CASCON First Decade High Impact Papers, 1999.

[38] **React Native · Learn once, write anywhere**. https://reactnative.dev/. Accessed 2023.

[39] **RFC 1738: Uniform Resource Locators (URL)**. https://www.rfc-editor.org/rfc/rfc1738. Accessed 2022.

[40] Riccardo Focardi and Flaminia L. Luccio and Heider A.M. Wahsheh. **Usable security for QR code**. In Proc. Journal of Information Security and Applications, 2019.

[41] **Risks of Using QRCodes and How to Mitigate it – Not as Safe as You Think**. https://www.computer.org/publications/tech-news/trends/qr-code-risks. Accessed 2022.

[42] **SHAREit - User Oriented Cross Platform & High-speed File Sharing Platform**. https://www.ushareit.com/product/shareit. Accessed 2022.

[43] **SHEIN - Fashion Shopping Online**. https://us.shein.com/. Accessed 2022.

[44] **Suishenban - Shanghai Municipal People's Government**. http://zwdt.sh.gov.cn/govPortals/column/download/application.html. Accessed 2022.

[45] Takayuki Ishihara and Michiharu Niimi. **Compatible 2D-Code Having Tamper Detection System with QR-Code**. In Proc. 10th International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP), 2014.

[46] **Taobao Web Site**. https://world.taobao.com/. Accessed 2022.

[47] Tongbo Luo and Hao Hao and Wenliang Du and Yifei Wang and Heng Yin. **Attacks on WebView in the Android System**. In Proc. 27th Annual Computer Security Applications Conference (ACSAC), 2011.

[48] Tongxin Li and Xueqiang Wang and Mingming Zha and Kai Chen and XiaoFeng Wang and Luyi Xing and Xiaolong Bai and Nan Zhang and Xinhui Han. **Unleashing the Walking Dead: Understanding Cross-App Remote Infections on Mobile WebViews**. In Proc. 24th ACM SIGSAC Conference on Computer and Communications Security (CCS), 2017.

[49] **Deadline Extended for App Updates Using UIWebView**. https://developer.apple.com/news/?id=edwud51q. Accessed 2022.

[50] **Allowing Apps and Websites to Link to Your Content**. https://developer.apple.com/documentation/xcode/allowing-apps-and-websites-to-link-to-your-content. Accessed 2022.

[51] **WebViewJavascriptBridge**. https://github.com/marcuswestin/WebViewJavascriptBridge. Accessed 2022.

[52] **Weex • A framework for building Mobile cross-platform UI**. https://github.com/alibaba/weex. Accessed 2023.

[53] **WhatsApp Web**. https://web.whatsapp.com/. Accessed 2022.

[54] Xing Jin and Xunchao Hu and Kailiang Ying and Wenliang Du and Heng Yin and Gautam Nagesh Peri. **Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation**. In Proc. ACM SIGSAC conference on computer and communications security (CCS), 2014.

[55] **Xposed**. https://github.com/rovo89/Xposed. Accessed 2022.

[56] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. TADOC: Text analytics directly on compression. The VLDB Journal, 30:163–188, 2021.

[57] Zihao Jin and Shuo Chen and Yang Chen and Haixin Duan and Jianjun Chen and Jianping Wu. **A Security Study about Electron Applications and a Programming Methodology to Tame DOM Functionalities**. In Proc. 30th Network and Distributed System Security Symposium (NDSS), 2023.

A Sample Code for RAH Registration

A.1 Android RAHs

Listing 3 Sample Java Code of JavascriptInterface

```java
// Register RAH in Java code
public class JSObject {
    @JavascriptInterface
    public void nativeMethod(String arg) {
        ...
    }
    ...
    webView.addJavascriptInterface(
        new JSObject(), "jsObject"
    );
}
```

Listing 4 Sample JavaScript Code of JavascriptInterface

```javascript
// Invoke RAH in JavaScript code
jsObject.nativeMethod(arg);
window.jsObject.nativeMethod(arg);
```

Listing 5 Sample Java Code of WebViewClient

```java
// Register in Java code
public class CustomizedClient extends WebViewClient {
    @Override
    public boolean onLoadResource(
        WebView view, String url
    ) {
        ...
    }
}
```

Listing 6 Sample JavaScript Code of WebViewClient

```javascript
// Trigger in JavaScript code
location.href = "<new URL>";
```

- **RaT-1**. As Listing 3 shows, in order to register this type of custom RAHs, the developers first add the annotation

@JavascriptInterface to the methods that will be registered as custom RAHs, then instantiate a Java object containing these methods and finally call the method addJavascriptInterface(Object obj, String name) to inject it into the JavaScript runtime in WebView. The first parameter of addJavascriptInterfacez is the Java object to inject and the second is the name of the injected JavaScript object.

As Listing 4 shows, in the web page opened in WebView, the injected object is a global variable in JavaScript and implicitly registered as a property of the window object. After injecting the object, the methods with the annotation @JavascriptInterface in the Java object are registered as the methods in the JavaScript object and can be directly invoked as any normal JavaScript methods (as the sample JavaScript code shows).

- **RaT-2**. As Listing 5 shows, in order to register this type of custom RAHs, the developers inherit the class WebViewClient and override the event handler methods in the class, and then instantiate the customized WebViewClient and call the method setWebViewClient(WebViewClient client) to set it in WebView. The WebViewClient instance can be retrieved by calling getWebViewClient method.

  As Listing 6 shows, if the web page opened in WebView triggers the events overridden in the customized WebViewClient, such as navigating to another URL by setting the value of location.href in the sample code, the corresponding custom event handler method will be invoked automatically, such as onLoadResource in the sample code. The specific new URL should be built according to the implementation of the customized handler methods.

A.2 iOS RAHs

We focused on the three most commonly used types of RAH in iOS apps:

- **RiT-1**. As Listing 7 shows, in order to register this type of custom RAHs, the developers will adopt the WKScriptMessageHandler protocol (like implementing an interface in other languages such as Java) and implement the userContentController:didReceiveScriptMessage: method defined in the protocol. Then the developers instantiate the WKWebViewConfiguration class as the configuration of WKWebView, and call the method addScriptMessageHandler:name: to add the customized handler to the configuration as the custom RAH. The parameter name is the name of the injected RAH. After that, the developers instantiate and initialize a WKWebView with the above configuration instance and

Listing 7 Sample Objective-C Code of WKScriptMessageHandler

```
// Register in Objective-C code
- (void)setupWKWebView{
    WKWebViewConfiguration *configuration =
        [[WKWebViewConfiguration alloc] init];
    configuration.userContentController =
        [[WKUserContentController alloc] init];
    [configuration.userContentController
        addScriptMessageHandler:self
        name:@"nativeMethod"];
    WKWebView *webView =
        [[WKWebView alloc]
            initWithFrame:self.view.frame
            configuration:configuration];
}
// Handler method defined in WKScriptMessageHandler
- (void)userContentController:
    (WKUserContentController*)userContentController
    didReceiveScriptMessage:(WKScriptMessage*)message {
    if ([message.name isEqualToString:@"nativeMethod"]) {
        ...
    }
}
```

Listing 8 Sample JavaScript Code for WKScriptMessageHandler

```
// Invoke in JavaScript code
window.webkit.messageHandlers.nativeMethod.postMessage();
```

Listing 9 Sample Objective-C Code of WebViewJavascriptBridge

```
self.bridge =
    [WebViewJavascriptBridge bridgeForWebView:webView];
[self.bridge
    registerHandler:@"nativeMethod"
    handler:
    ^(id data, WVJBResponseCallback responseCallback) {
        ...
    }
];
```

Listing 10 Sample Objective-C Code of DSBridge

```
@implementation JsObject
- (NSString *) nativeMethod:(NSString *) msg
{
    ...
}
@end
DWKWebView* dwebview =
    [[DWKWebView alloc] initWithFrame:bounds];
[dwebview addJavascriptObject:[[JsObject alloc] init]
    namespace:nil];
```

Listing 11 Sample JavaSrcript Code of DSBridge

```
var dsBridge=require("dsbridge");
var str=dsBridge.call("nativeMethod","arg");
```

then the RAHs can be invoked directly from the web page in the WKWebView.

As Listing 8 shows, in the web page opened in WKWebView, the RAHs can be invoked through the window.webkit.messageHandlers object. The property nativeMethod is as same as the name registered above.

- **RiT-2**. As Listing 9 shows, in order to register this type of custom RAHs, the developers will call the bridgeForWebView: method to initiate WebViewJavascriptBridge and set the instance as the property navigationDelegate of WKWebView and then call the registerHandler:handler: method to register an Objective-C block (like a closure or lambda in other languages) as an RAH. The first parameter is the name of the RAH used to invoke it from the web page. All the registered handlers will be stored in a variable named messageHandlers and can be retrieved by the method [_base messageHandlers]. As the navigationDelegate of WKWebView, the WebViewJavascriptBridge instance can intercept the navigation and requests in the WKWebView.

  In WebViewJavascriptBridge, the RAHs will not be injected into WKWebView when it is created. As Listing 1 shows, only when the URL specified by the developers is requested in the web page using iframe, intercepted and verified by the WebViewJavascriptBridge instance, the method injectJavascriptFile will be invoked to inject the bridge object to the JavaScript runtime in WKWebView. The value of WVJBIframe.src is the specific URL to trigger the injection which can be customized in the app and the same URL must be used in the web page.

- **RiT-3**. As Listing 10 shows, in order to register this type of custom RAHs, the developers will instance the DWKWebView class which is a sub class of WKWebView instead of WKWebView itself and call the addJavascriptObject:namespace: method to add an Objective-C object as RAH to DWKWebView. All the added RAHs will be stored in the member variable javaScriptNamespaceInterfaces of DWKWebView.

  As Listing 11 shows, in the web page opened in DWKWebView, the web page require the JavaScript library of DSBridge and use it to invoke the custom RAHs. The names used to invoked RAHs are as same as the method names defined in the Objective-C object.